OpenGL ES Tuning and Techniques

Session 312

These are confidential sessions—please refrain from streaming, blogging, or taking pictures.

OpenGL ES Tuning and Techniques

John Rosasco and Benj Lipchak

Session Overview

Topics

- Hardware architecture
  - iPhone and iPod touch
  - iPhone 3G S
- Top 10 performance optimizations
- Finding bottlenecks

JDR: overlay hardware archs and ES versions in build if possible for comparison

JDR: speaker note - note that hardware differences will be noted as each of the optimization methods are described.

iPhone and iPod Touch Hardware Architecture

iPhone and iPod Hardware Architecture

Imagination PowerVR MBX-lite

- Tile-based deferred renderer
- Supports OpenGL ES 1.1 only
- 24 MB limit on texture and surface memory

JDR: fixed function only
JDR: compare to stream renderer
JDR: describe surfaces orally

Get terminology for existing install base from Allan.

iPhone 3G S Hardware Architecture

Based on PowerVR SGX

- Tile-based deferred renderer
- Supports both OpenGL ES 1.1 and 2.0
- Fully programmable hardware shading
- Higher memory bandwidth
- No hard limit on texture and surface memory
- Hardware VBO support
- Stencil buffer support

JDR: explain that VBOs are supported on MBX/ES1.1 but are not higher performance than standard client-side vertex arrays.

Tile-Based Deferred Rendering

Comparing TBDR and Stream Rendering

A Stream Rendering GPU draws one frame

```
while (more drawing commands)
  {
     while (more state changes)
     {
         accumulate state change  // Any non-draw command
     }

    draw command                   // glDrawArrays, glDrawElements ...
 }
```

```
display frame on screen          // [ctxt presentRenderBuffer: ...];
```

Comparing TBDR and Stream Rendering

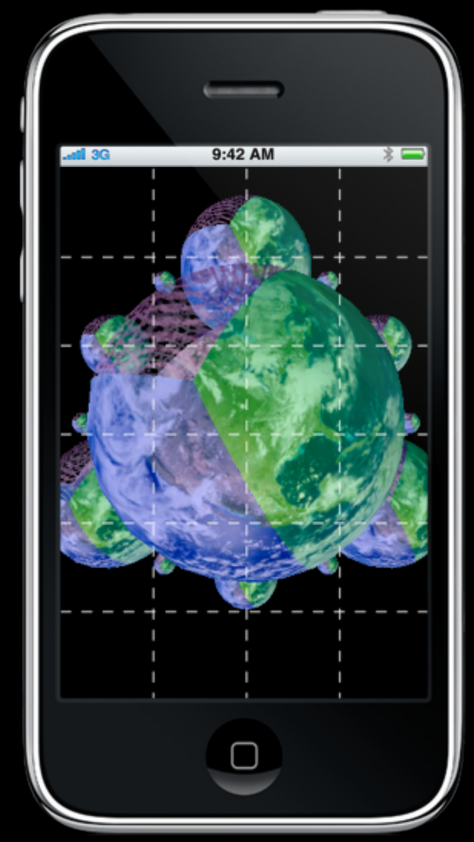
An iPhone Tile-Based Deferred Rendering GPU draws a frame

```
while (more drawing commands in frame)
{
  cache drawing command          // All GL ES Commands
}
```

```
divide frame data into tiles
create a list of drawing commands for each tile
```

```
foreach (tile)
{
  load color, depth, and stencil buffers
  determine visibilty of pixels
  execute fragment shader for visible pixels
  store color, depth and stencil buffers
}
```

```
display frame on screen          // [ctxt presentRenderBuffer: ...];
```



Implementation Details

Why TBDR matters to your application

JDR:
(1) Fetch only visible texels and writes only visible pixels.
(2) All 3 of these bullets can force the TBDR to break a frame and do an additional expensive tiling pass over the framebuffer.
(3) Schedule BindFramebuffer calls at the beginning, ReadPixels at the end.
(4) For texture update commands, try to avoid them if possible but if not schedule them before the first or after

- TBDR uses memory more efficiently
  - GPU reads only visible texels
  - GPU writes only visible fragments
- But…
  - Operations that depend on previous GL commands can be expensive
  - Example: Carefully schedule operations such as
    - glBindFramebufferOES
    - glReadPixels
    - glTexSubImage / glCopyTexImage / glCopyTexSubImage

Implementation Details

TBDR suggested frame architecture

```
// BEGINNING OF FRAME - Preferred time to do texture updates - ** avoid if possible **
glTexSubImage() or glCopyTexImage() or glCopyTexSubImage()
// Preferred time to bind your FBO
glBindFramebufferOES()

Other GL commands in this frame ...
Other GL commands in this frame ...

// Last Draw Command in Frame
glDrawElements() or glDrawArrays()
// Now ok to do a glReadPixels if required
glReadPixels()
// END OF FRAME
[ctx presentRenderBuffer: GL_RENDERBUFFER]
```

JDR:
(1) Fetch only visible texels and writes only visible pixels.

(2) All 3 of these bullets can force the TBDR to break a frame and do an additional expensive tiling pass over the framebuffer.

(3) Schedule BindFramebuffer calls at the beginning, ReadPixels at the end.

(4) For texture update commands, try to avoid them if possible but if not schedule them before the first or after the last draw command of

The Top 10 List

Key Optimization Strategies

The Top 10 List

Key Optimization Strategies

1. Avoid alpha test and "discard"
2. Don’t depth-sort opaque surfaces
3. Batch draws and minimize state changes
4. Use proper vertex data management
5. Use proper texture data management
6. Minimize memory footprint
7. Use GL to draw landscaped content
8. Make your GL layer opaque
9. Draw only when needed
10. Write efficient shaders*

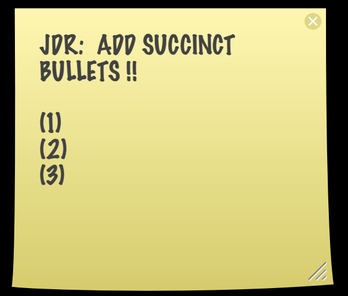
The Top 10 List

Key Optimization Strategies

1. Avoid alpha test and "discard"
2. Don’t depth-sort opaque surfaces
3. Batch draws and minimize state changes
4. Use proper vertex data management
5. Use proper texture data management
6. Minimize memory footprint
7. Use GL to draw landscaped content
8. Make your GL layer opaque
9. Draw only when needed
10. Write efficient shaders*

Avoid Alpha Test and “discard”

Tip #1



- Alpha Test is costly
  - Defeats early depth testing and hidden surface removal in the TBDR
  - Opt for framebuffer blending instead
- Use of “discard” in shaders should be avoided
  - Discard results in an expensive visibility pass in the GPU

The Top 10 List

Key Optimization Strategies

1. Avoid alpha test and "discard"
2. Don’t depth-sort opaque surfaces
3. Batch draws and minimize state changes
4. Use proper vertex data management
5. Use proper texture data management
6. Minimize memory footprint
7. Use GL to draw landscaped content
8. Make your GL layer opaque
9. Draw only when needed
10. Write efficient shaders*

Don’t Depth-sort Opaque Surfaces

Tip #2

- TBDR makes depth sorting redundant
- Wastes precious CPU cycles
- Do follow these ordering rules
  - Render opaque objects first
  - Render objects with discard second
  - Render sorted, blended objects last

JDR:
(1) Opaque objects allow the GPU to efficiently perform large granularity culling.

(2) Shaders with discard force the GPU to do an expensive pass over the data.

(3) Blended object depend on the framebuffer destination for correct results and must be rendered last.

The Top 10 List

Key Optimization Strategies

1. Avoid alpha test and "discard"
2. Don’t depth-sort opaque surfaces
3. Batch draws and minimize state changes
4. Use proper vertex data management
5. Use proper texture data management
6. Minimize memory footprint
7. Use GL to draw landscaped content
8. Make your GL layer opaque
9. Draw only when needed
10. Write efficient shaders*

Batch Drawing to Minimize State Changes

Tip #3

JDR:

(1) OOP Programming
(2) App Domain vs Gfx Pipeline Domain
(3) 2 Examples

- Batch as much work as possible into a single draw call

```
#if BATCHED
    glBindBuffer(GL_ARRRAY_BUFFER, treeVBO);
    glBindTexture(GL_TEXTURE_2D, leafTex);
    glDrawArrays(GL_TRIANGLES, leavesStart, leavesVtx);
#else
    // Leaf draw method
     - (void) draw {
         glBindBuffer(GL_ARRRAY_BUFFER, self.vbo);
         glBindTexture(GL_TEXTURE_2D, self.tex);
         glDrawArrays(GL_TRIANGLES, 0, self.vtx);
     }

     // leaf:draw called elsewhere in code ...
     for(i = 0; i < leaves; i++) {
        [[tree getLeaf: i] draw];
     }
#endif
```

The Top 10 List

Key Optimization Strategies

1. Avoid alpha test and "discard"
2. Don’t depth-sort opaque surfaces
3. Batch draws and minimize state changes
4. Use proper vertex data management
5. Use proper texture data management
6. Minimize memory footprint
7. Use GL to draw landscaped content
8. Make your GL layer opaque
9. Draw only when needed
10. Write efficient shaders*

The Top 10 List

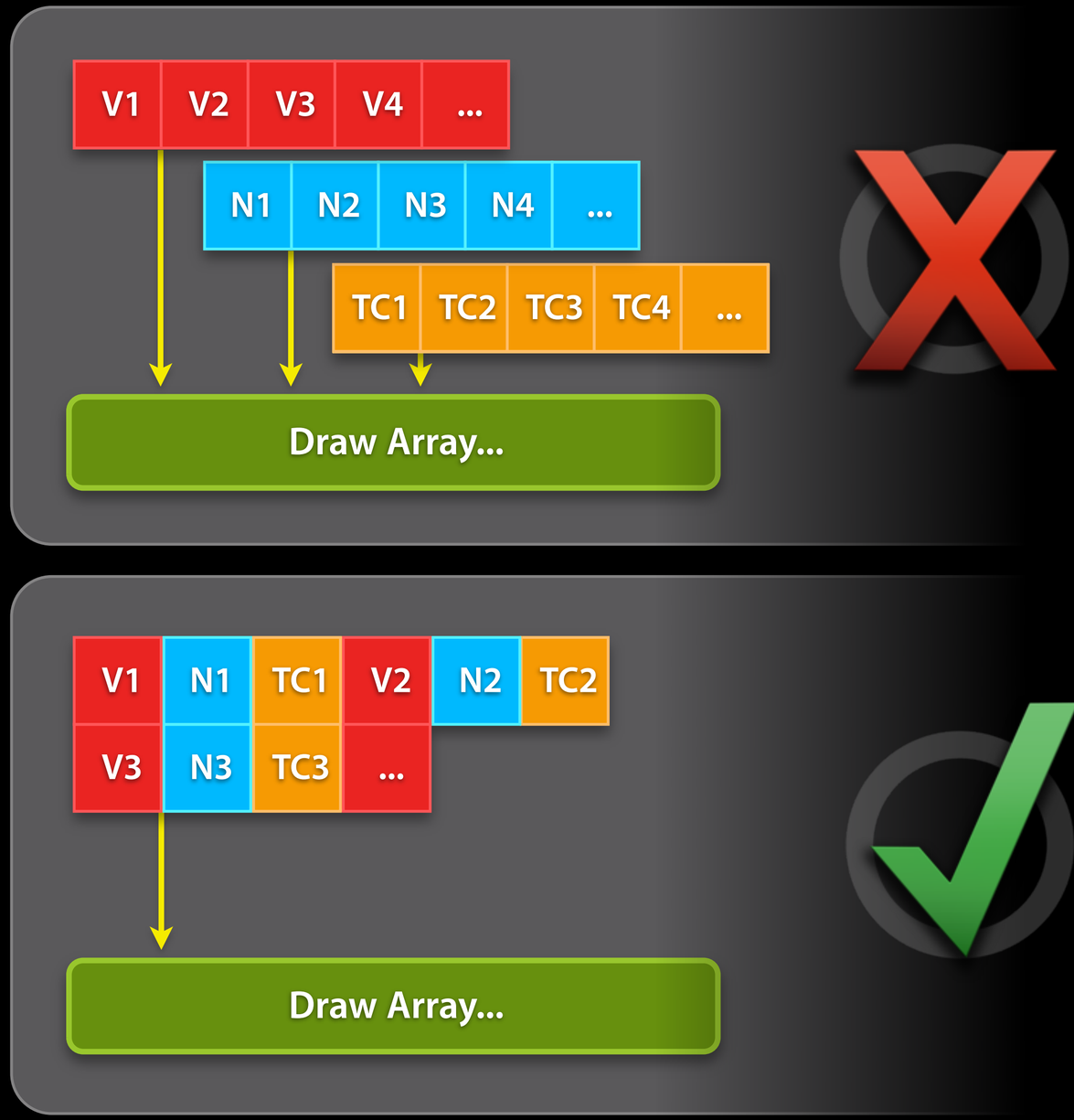
Key Optimization Strategies

1. Avoid alpha test and "discard"
2. Don’t depth-sort opaque surfaces
3. Batch draws and minimize state changes
4. Use proper vertex data management
5. Use proper texture data management
6. Minimize memory footprint
7. Use GL to draw landscaped content
8. Make your GL layer opaque
9. Draw only when needed
10. Write efficient shaders*

Use Proper Vertex Data Management

Tip #4

- Use VBOs
- Align each vertex attribute and their strides on 4 byte boundaries
- Use smaller data types
- Interleave vertex data



Use Proper Vertex Data Management

Tip #4 — Code sample

```
#if FASTER_SMALLER_INTERLEAVED_ALIGNED_VERTICES
struct interleaved_s {
    GLshort vtx[4], norm[4];  // 8 bytes + 8 bytes == 16 bytes per vertex
} interleaved[NUM_VERTS];


glVertexPointer(4, GL_SHORT, sizeof(struct interleaved_s), &interleaved[0].vtx);
glNormalPointer(GL_SHORT, sizeof(struct interleaved_s), &interleaved[0].norm);
#else // Slower, Bigger, Non-interleaved, Unaligned Vertices
GLfloat vtx[NUM_VERTS][3];
GLshort norm[NUM_VERTS][3];   // 12 bytes + 6 bytes == 18 bytes per vertex


glVertexPointer(3, GL_FLOAT, 0, vtx);
glNormalPointer(GL_SHORT, 0, norm);  // stride == 6 bytes so not aligned
#endif
```

The Top 10 List

Key Optimization Strategies

1. Avoid alpha test and "discard"
2. Don’t depth-sort opaque surfaces
3. Batch draws and minimize state changes
4. Use proper vertex data management
5. Use proper texture data management
6. Minimize memory footprint
7. Use GL to draw landscaped content
8. Make your GL layer opaque
9. Draw only when needed
10. Write efficient shaders*

Use Proper Texture Data Management

Tip #5

- Use compressed textures
- Use mipmapping
  - Bi-linear filtering unless tri-linear required
- Use texture atlases

JDR:
(1) compressed textures -> performance & quality

(2) Atlases and Mipmapping

(3) Frame data aggregation of TBDR requires copies to be made when textures are updated.

- Avoid mid-frame texture updates
  - glTexSubImage()
  - glCopyTexImage()
  - glCopyTexSubImage()

Use Proper Texture Data Managment

Tip #5 — Code sample

```
// Use Compressed, Mipmapped Textures
struct texLevel_s {
    GLsizei width, height, size;
    GLvoid *data;
} texLevels[NUM_LEVELS];

glBindTexture(GL_TEXTURE_2D, compressedMipmappedTexID);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);

Set texture mag filter and wrap modes ...

for(i = 0; i < NUM_LEVELS; i++)
    glCompressedTexImage2D(GL_TEXTURE_2D, i, GL_COMPRESSED_RGBA_PVRTC_4BPPV1_IMG,
        texLevel[i].width, texLevel[i].height, 0, texLevel[i].size, texLevel[i].data);
```

The Top 10 List

Key Optimization Strategies

1. Avoid alpha test and "discard"
2. Don’t depth-sort opaque surfaces
3. Batch draws and minimize state changes
4. Use proper vertex data management
5. Use proper texture data management
6. Minimize memory footprint
7. Use GL to draw landscaped content
8. Make your GL layer opaque
9. Draw only when needed
10. Write efficient shaders*

Minimize Memory Footprint

Tip #6

- Texture format
  - Compressed
  - Mipmapped
- Vertex format
  - Minimal size
- Avoid mid-frame texture updates
- Only allocate depth buffer if needed

The Top 10 List

Key Optimization Strategies

1. Avoid alpha test and "discard"
2. Don’t depth-sort opaque surfaces
3. Batch draws and minimize state changes
4. Use proper vertex data management
5. Use proper texture data management
6. Minimize memory footprint
7. Use GL to draw landscaped content
8. Make your GL layer opaque
9. Draw only when needed
10. Write efficient shaders*

The Top 10 List

Key Optimization Strategies

1. Avoid alpha test and "discard"
2. Don’t depth-sort opaque surfaces
3. Batch draws and minimize state changes
4. Use proper vertex data management
5. Use proper texture data management
6. Minimize memory footprint
7. Use GL to draw landscaped content
8. Make your GL layer opaque
9. Draw only when needed
10. Write efficient shaders*

Use GL to Draw Landscaped Content

Tip #7

JDR: Add code snippet showing how to reorient the status bar.

- Prefer GL viewport and model transforms over using CALayer transforms to draw landscaped content
  - Swap the width and height on the viewport sizing
  - Add a model-space rotation around k (0.0, 0.0, 1.0) to keep model upright

The Top 10 List

Key Optimization Strategies

1. Avoid alpha test and "discard"
2. Don’t depth-sort opaque surfaces
3. Batch draws and minimize state changes
4. Use proper vertex data management
5. Use proper texture data management
6. Minimize memory footprint
7. Use GL to draw landscaped content
8. Make your GL layer opaque
9. Draw only when needed
10. Write efficient shaders*

Make Your GL Layer Opaque

Tip #8

- Avoid compositing EAGL layers over other Core Animation layers
  - OK to composite 2D content over GL content

```
// From EAGLView - UIView subclass for GL ES drawing
- (id) initWithCoder:(NSCoder *) coder
{
CAEAGLLayer *eaglLayer = (CAEAGLLayer *) self.layer;
eaglLayer.opaque = YES;

remainder of EAGLView initWithCoder method ...
}
```

The Top 10 List

Key Optimization Strategies

1. Avoid alpha test and "discard"
2. Don’t depth-sort opaque surfaces
3. Batch draws and minimize state changes
4. Use proper vertex data management
5. Use proper texture data management
6. Minimize memory footprint
7. Use GL to draw landscaped content
8. Make your GL layer opaque
9. Draw only when needed
10. Write efficient shaders*

Draw Only When Needed

Tip #9

- Limit frame rate

```
[NSTimer scheduledTimerWithTimeInterval: animationInterval
    target:self selector:@selector(drawView) userInfo:nil repeats:YES];
```

- Stop drawing for a scene that’s not changing

```
- (void) stopAnimation
{
    [self.animationTimer invalidate];
    self.animationTimer = nil;
}
```

The Top 10 List

Key Optimization Strategies

1. Avoid alpha test and "discard"
2. Don’t depth-sort opaque surfaces
3. Batch draws and minimize state changes
4. Use proper vertex data management
5. Use proper texture data management
6. Minimize memory footprint
7. Use GL to draw landscaped content
8. Make your GL layer opaque
9. Draw only when needed
10. Write efficient shaders*

Write Efficient Shaders*

Tip #10

- General tips
- Respect device limits
- Hoist computation up and out
- Efficient texture fetching
- Make empirical precision choices

Write Efficient Shaders*

Tip #10

- General tips
- Respect device limits
- Hoist computation up and out
- Efficient texture fetching
- Make empirical precision choices

Shaders

Tip #10 — General Tips

- Always create/compile/link at initialization time
- Unroll small loops
- Use static array access if possible
- Branching has cost

Write Efficient Shaders*

Tip #10

- General tips
- Respect device limits
- Hoist computation up and out
- Efficient texture fetching
- Make empirical precision choices

Shaders

Tip #10 — Respect device limits

JDR: briefly describe packing rules orally

- Query the limits on the qualified variables of shaders
  - Uniforms, attributes, varyings
  - Conserve where possible

```
// Uniform limit
glGetIntegerv(GL_MAX_VERTEX_UNIFORM_VECTORS, &maxVtxUniforms);
glGetIntegerv(GL_MAX_FRAGMENT_UNIFORM_VECTORS, &maxFragUniforms);
// Vertex attrib limit
glGetIntegerv(GL_MAX_VERTEX_ATTRIBS, &maxVtxAttribs);
// Varying limit
glGetIntegerv(GL_MAX_VARYING_VECTORS, &maxVaryingVectors);
```

Write Efficient Shaders*

Tip #10

- General tips
- Respect device limits
- Hoist computation up and out
- Efficient texture fetching
- Make empirical precision choices

Shaders

Tip #10 — Hoist computation up and out

- Note the number of operations that will occur in your shader!
  - At least per-vertex if not per-fragment
- Pick the right stage for an operation
  - Prefer per-vertex ops to per-fragment ops
  - Prefer uniforms or simple CPU ops to per-vertex ops

Write Efficient Shaders*

Tip #10

- General tips
- Respect device limits
- Hoist computation up and out
- Efficient texture fetching
- Make empirical precision choices

Shaders

Tip #10 — Efficient texture fetching

- Texture fetches
  - Issue them early to improve parallelism
- Dependent texture reads
  - Avoid modifying texture coordinates in your fragment shader

JDR: sgx/mbx have distinct texture fetching units
JDR: describe dependent texture reads and the costs of modulating texture coordinates within a shader (no way to amortize the latency and time required to fetch texels).

```
uniform sampler2D tex0;
uniform mediump float bias;
varying mediump vec2 TexCoord0;

void main(void) {
    mediump vec2 tmp = TexCoord0 + bias;  // AVOID MODIFYING TEX COORDS PRIOR TO SAMPLING
    gl_FragColor = texture2D(tex0, tmp);
}
```

JDR: scalar vs vector operations. Check Imagination docs that Richard circulated.

Write Efficient Shaders*

Tip #10

- General tips
- Respect device limits
- Hoist computation up and out
- Efficient texture fetching
- Make empirical precision choices

Shaders

Tip #10 — Make empirical precision choices

- lowp — Range of -2 to +2, 8 bits of precision
- mediump — Half precision
- highp — Single precision

```
varying lowp Color;
```

- Be empirical about your precision qualifiers!

The Top 10 List

Key Optimization Strategies

1. Avoid alpha test and discard
2. Don’t depth-sort opaque surfaces
3. Batch draws and minimize state changes
4. Use proper vertex data management
5. Use proper texture data management
6. Minimize memory footprint
7. Use GL to draw landscaped content
8. Make your GL layer opaque
9. Draw only when needed
10. Write efficient shaders*

Finding Bottlenecks

Finding Bottlenecks

Where is my performance going ?

- CPU limited
- Vertex processing limited
- Texture bandwidth limited
- Fragment shading limited
- Compositing limited

JDR:

(1) Finding bottlenecks efficiently is all about making the fewest changes in the source code that will cause the largest change in the frame rate.

CPU Limited

Shark

- Use shark to evaluate logical hot spots in your application source code
  - If the CPU is pegged at 100%, chances are you're CPU limited

Vertex Processing Limited

Storms of large vertices

- Bandwidth limited
  - Use smaller sized vertex attribute types
  - Convert to VBOs
  - Align vertex data
- Shading limited
  - Reduce vertex shading to pass-thru and check FPS

Texture Bandwidth Limited

Tarp-sized textures

- Use compressed, mipmapped textures
- Use LOD controls to determine if you are texture bandwidth limited
  - ES1
    - EXT_texture_lod_bias
      - glTexEnvf(GL_TEXTURE_FILTER_CONTROL_EXT, GL_TEXTURE_LOD_BIAS_EXT, biasValue);
  - ES2
    - texture2D(sampler2D sampler, vec2 coord, float bias);
- Limit or remove texture fetches in fragment shader

Fragment Shading Limited

Too many cooks spoil the fragment

- Reduce window size and watch FPS
- Create a pass-thru fragment shader and watch FPS
- Successively simplify fragment shader
  - Reduce branching
  - Reduce dynamic array accesses
  - Shorten shader

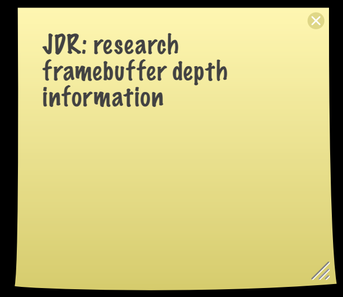
Compositing Limited

Core Animation

- Review all Core Animation code
  - Disable compositing
  - Remove layer transforms

Summary

Recap of todays session



- Start by identifying bottlenecks
- Batch drawing state
- Choose appropriate data types, formats, and precision
  - Conserve memory where possible
- Account for variable declarations in shaders
  - Varyings, uniforms, attributes, and locals
- Know how TBDR differs from stream rendering

Related Sessions

Performance Tuning with Shark on Mac and iPhone	North Beach Friday 9:00AM

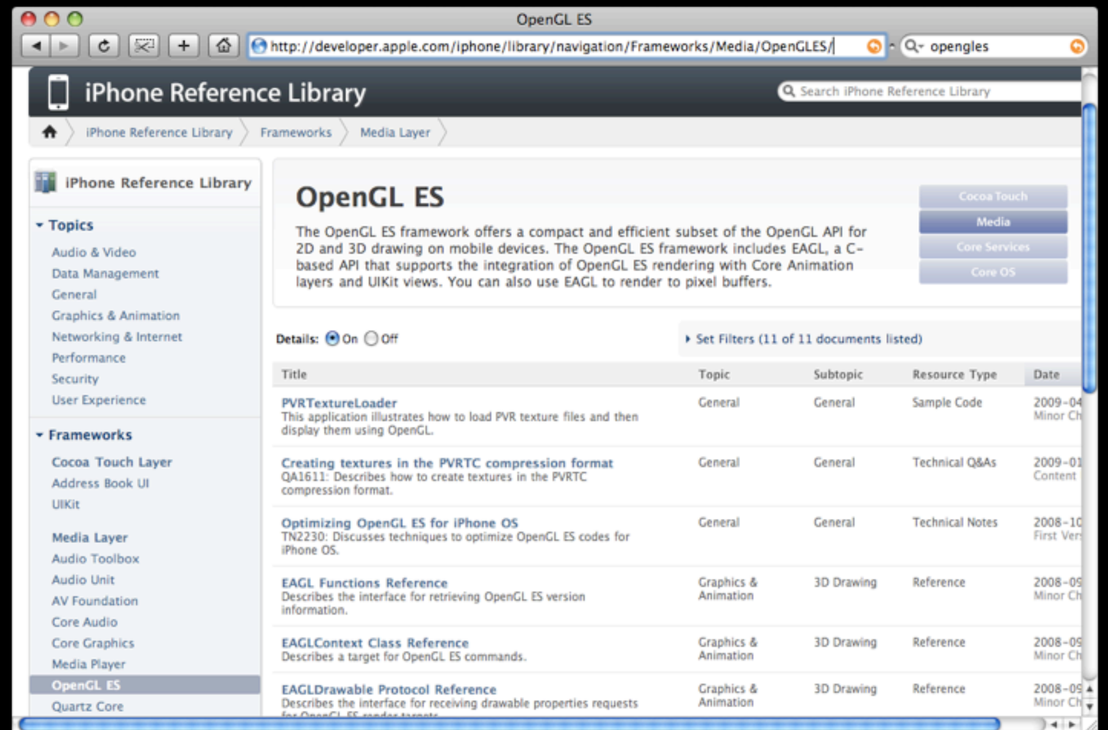
Labs

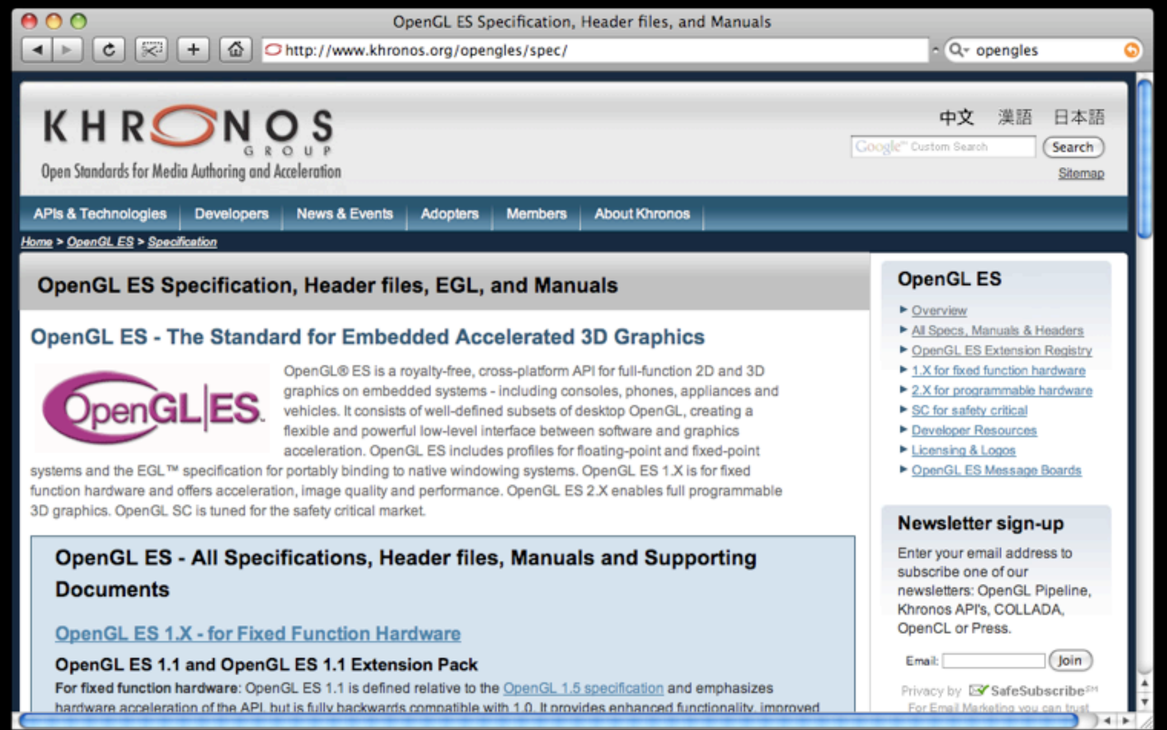
Game Development for iPhone	iPhone Lab C Thursday 9:00AM
OpenGL ES Lab	Graphics and Media Lab A Thursday 12:00PM

Documentation

http://developer.apple.com/iphone/library/navigation/Frameworks/Media/OpenGLES/

http://www.khronos.org/opengles/spec/





More Information

Allan Schaffer
Graphics Technology Evangelist
aschaffer@apple.com

Q&A

WWDC Session Survey

You can rate your sessions on the WWDC Attendee Site.

Survey forms are located on each session description page at:

developer.apple.com/wwdc/attendee

WWDC09